# 目录

[目录](#5237-1611023227637)

[Hash（散列函数）](#8612-1611023748044)

[常用HASH函数](#5046-1611023415927)

[处理冲突方法](#5575-1611023498142)

[HashMap](#1377-1611023770376)

# Hash（散列函数）

Hash，一般翻译做散列或音译为哈希，是把任意长度的输入，通过散列算法变换成固定长度的输出，该输出就是散列值，又称hash值(**原始数据映射后的二进制串**)。

**通过构造性能良好的哈希函数，可以减少冲突，但一般不可能完全避免冲突，因此解决冲突是哈希法的另一个关键问题。**

**Hash的特点：**

1.从hash值不可以反向推导出原始的数据

2.输入数据的微小变化会得到完全不同的hash值，相同的数据会得到相同的值

3.哈希算法的执行效率要高效，长的文本也能快速地计算出哈希值

4.hash算法的冲突概率要小

由于hash的原理是将输入空间的值映射成hash空间内，而hash值的空间远小于输入的空间。

根据抽屉原理，一定会存在不同的输入被映射成相同输出的情况。

**抽屉原理**：桌上有十个苹果，要把这十个苹果放到九个抽屉里，无论怎样放，我们会发现至少会有一个抽屉里面放不少于两个苹果。 这一现象就是我们所说的“抽屉原理”。

# 常用HASH函数

**1．直接寻址法**。取关键字或关键字的某个线性函数值为散列地址。即H(key)=key或H(key) = a·key + b，其中a和b为常数（这种散列函数叫做自身函数）

**2．数字分析法**。分析一组数据，比如一组员工的出生年月日，这时我们发现出生年月日的前几位数字大体相同，这样的话，出现冲突的几率就会很大，但是我们发现年月日的后几位表示月份和具体日期的数字差别很大，如果用后面的数字来构成散列地址，则冲突的几率会明显降低。因此数字分析法就是找出数字的规律，尽可能利用这些数据来构造冲突几率较低的散列地址。

**3．平方取中法**。取关键字平方后的中间几位作为散列地址。

**4**．**折叠法**。将关键字分割成位数相同的几部分，最后一部分位数可以不同，然后取这几部分的叠加和（去除进位）作为散列地址。

**5．随机数法**。选择一随机函数，取关键字作为随机函数的种子生成随机值作为散列地址，通常用于关键字长度不同的场合。

**6．除留余数法**。取关键字被某个不大于散列表表长m的数p除后所得的余数为散列地址。即 H(key) = key MOD p,p<=m。不仅可以对关键字直接取模，也可在折叠、平方取中等运算之后取模。对p的选择很重要，一般取素数或m，若p选的不好，容易产生碰撞。

# 处理冲突方法

**1．开放寻址法**；Hi=(H(key) + di) MOD m,i=1,2,…，k(k<=m-1)，其中H(key)为散列函数，m为散列表长，di为增量序列，可有下列三种取法：

1)． di=1,2,3,…，m-1，称线性探测再散列；

2)． di=1^2,-1^2,2^2,-2^2,3^2,…，±k^2,(k<=m/2)称二次探测再散列；

3)． di=伪随机数序列，称伪随机探测再散列。

**2． 再散列法**：Hi=RHi(key),i=1,2,…，k RHi均是不同的散列函数，即在同义词产生地址冲突时计算另一个散列函数地址，直到冲突不再发生，这种方法不易产生“聚集”，但增加了计算时间。

**3．** **链地址法(拉链法)**

**4．** **建立一个公共溢出区**

# HashMap

HashMap是Java程序员使用频率最高的用于映射(键值对)处理的数据类型。

基于哈希表的 Map 接口的实现.

hashMap 的数据初始容量默认16。

hashMap 的数组容量必须是2的N次幂。

**HashMap的数组容量为什么是2的N次幂？**

**简化答案** ：因为在对插入的元素定位数组位置时，把hashCode和容量进行按位与计算。使用2的次幂进行计算时可以减少碰撞的发生，并且让散列尽可能的均匀分布。

**详细答案**：数组的索引值是0到N-1，而hashCode的计算范围是42亿，也就是所有对象的hashCode有42亿种可能，当length总是 2 的n次方时，h& (length-1)运算等价于对length取模，也就是h%length，但是&比%具有更高的效率。我们要把它丢到这16的长度的数组里面时，我们应该怎么做？怎么将42亿种可能变为0到N-1种可能，通常我们会想到用取模的方法来计算,但是它有两个缺点：1、负数求余还是负数 2、 性能较慢（对比位运算）。HasnMap使用**(table.length - 1) & hash ,** 那为什么是2的N次幂呢？ 2的N次幂转为二进制是1后面全是0, 比如16的二进制为10000，减1就等于1111，通过&运算，获得的结果分布比较均匀，减少hash碰撞，避免形成链表的结构，使得查询效率降低！因此，扩容也是按2倍扩容的。

 HashMap的容量为什么是2的n次幂，和这个(n - 1) & hash的计算方法有着千丝万缕的关系。符号&是按位与的计算，这是位运算，计算机能直接运算，特别高效，按位与&的计算方法是，只有当对应位置的数据都为1时，运算结果也为1，当HashMap的容量是2的n次幂时，(n-1)的2进制也就是1111111\*\*\*111这样形式的，这样与添加元素的hash值进行位运算时，能够充分的散列，使得添加的元素均匀分布在HashMap的每个位置上，减少hash碰撞。

**hashMap扩容**

1. hashMap扩容的条件有两个，一个是hashMap.size() >= length \* loadFactor ,还有另一个条件：当前要存放的下标位置不为空，如果为空，是不会扩容的。

2. 数组一旦确定长度是无法改变的，无法自动扩容，必须new出来一个新的数组来完成扩容操作。

3. 链表扩容的时候，会计算出两个index下标位置，一个是原下标，一个是原下标+原数组长度=新下标。（*old table.length + 当前index）.*

4. 红黑树扩容的时候，会先拆分成高位和低位两个链表，如果全部拆分到低位链表，则将整颗红黑树移动到新下标位置，如果拆分为高低两个链表，则判断是否满足链转树的条件，

第一次put时会初始化数组，其容量变为**不小于指定容量的2的幂数。**

如果不是第一次扩容，则 ​新容量= 旧容量 x 2​。

JDK1.7的阈值在第一次扩容之后计算出阈值，在非第一扩容之后，按照原公式重新计算阈值。

JDK1.8的阈值在第一次扩容之后计算出阈值，之后都是对​阈值 x 2。

HashMap1.8 首次插入数据时，先发生resize扩容再插入数据，之后每当插入的数据个数达到threshold时就会发生resize，此时是先插入数据再resize

jdk1.7中当哈希表为空时，会先调用inflateTable()初始化一个数组；而1.8则是直接调用resize()扩容;

**JDK7的元素迁移**

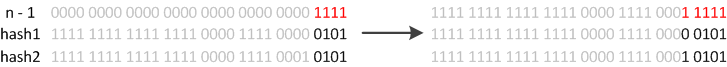
JDK7中，HashMap的内部数据保存的都是链表。因此逻辑相对简单：在准备好新的数组后，map会遍历数组的每个“桶”，然后遍历桶中的每个Entity，重新计算其hash值，找到新数组中的对应位置，以头插法插入新的链表。

这里有几个注意点：

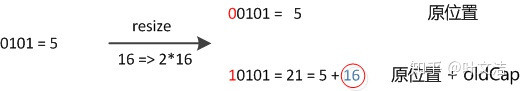
* 因为是头插法，因此新旧链表的元素位置会发生转置现象。
* 元素迁移的过程中在多线程情境下有可能会触发死循环（无限进行链表反转）。

**JDK8的元素迁移**

JDK8则因为巧妙的设计，性能有了大大的提升：由于数组的容量是以2的幂次方扩容的，那么一个Entity在扩容时，新的位置要么在原位置，要么在原长度+原位置的位置。原因如下图：



数组长度变为原来的2倍，表现在二进制上就是前面多了一个1。此时，各个元素重新hash&新的length之后，恰好出现一个现象：最高位是0则坐标不变，最高位是1则坐标变为“10000+原坐标”，即“原长度+原坐标”。如下图：



因此，在扩容时，不需要重新计算元素的hash了，只需要判断原hash值的最高位是1还是0就好了。

可以根据 e.hash & oldCap 的结果来判断，如果是0，说明位置没有发生变化，如果不为0（刚好是原数组长度），说明位置发生了变化，而且新的位置=老的位置+老的数组长度。

JDK8的HashMap还有以下细节：

* JDK8在迁移元素时是尾插法，不会出现链表反转，也就不会出现死循环。
* 如果某个桶内的元素超过8个，则会将链表转化成红黑树，加快数据查询效率。

**HashMap链表转红黑树：**

1. 当链表长度超过8，并且hashMap的数组长度大于等于64，则触发链转树。

2. 链转树的时候，先将链表转为一条由TreeNode组成的双向链表。（next和prev，prev指向上一个元素）。然后开始树化，由第一个treeNode开始遍历，生成一颗红黑树，一个一个元素往树里面插入。

**为什么链转树阈值是8？**

答：和hashcode碰撞次数的泊松分布有关，主要是为了寻找一种时间和空间的平衡。之所以是8，是因为Java的源码贡献者在进行大量实验发现，hash碰撞发生8次的概率已经降低到了0.00000006，几乎为不可能事件，如果真的碰撞发生了8次，那么这个时候说明由于元素本身和hash函数的原因，此次操作的hash碰撞的可能性非常大了，后序可能还会继续发生hash碰撞。所以，这个时候，就应该将链表转换为红黑树了，也就是为什么链表转红黑树的阈值是8。

**红黑树转链表的阈值为什么是6？**

答：主要为了一个过渡，避免链表和红黑树之间频繁的转换。如果也将该阈值设置于8，那么当hash碰撞在8时，会反生链表和红黑树的不停相互激荡转换，白白浪费资源。

// 以下为网友提供，挺精辟的！

